

CS0012 Introduction to Computing for the Humanities

Project 3

Overview

In this project, you will use your understanding of object-oriented programming to recreate a famous study on racial segregation. Using objects to represent “people” and “neighborhoods” will allow you to create a computer simulation mirroring the original study efficiently and effectively.

Background

For an idea of what you’ll be tackling with this project, you can take a look at [Parable of the Polygons](#). This interactive demonstration shows how small individual amounts of “shapist” bias can result in large amounts of segregation in the aggregate.

Per the acknowledgments at the bottom of the website, this interactive parable is based on a 1971 article by Thomas Schelling that has been cited over 4,000 times (according to Google Scholar). Using pennies and nickels as stand-ins for race, his model tried to isolate and study one factor that could contribute to racial segregation: the scaling up of individual decisions, made without regard to the actions of others. In brief, he found that there would be a dramatic over-satisfaction of demands: from an initial state of near-integration (46% of same-coin neighbors, on average), moving coins to achieve a minimum of 50% same-coin neighbors led to an equilibrium state with an average 80% same-coin neighbors. In short: the quest for an evenly mixed neighborhood led to an extremely segregated one. This was true under several possible variations and initial conditions, and in a 1991 study the result was found to be consistent with real-world survey data, from multiple cities.

In this project, you will write a program to simulate many iterations on Schelling’s original model. Schelling’s model used the following setup:

The collection of “neighborhoods” is represented as a grid with 13 rows and 16 columns, and with coins distributed at random among the spaces, leaving some spaces blank (about 25-30% blank spaces seemed to work well). Start with an equal number of the two coins, i.e. each being half the number of non-blank spaces. Let the “neighborhood” of any one coin equal the surrounding eight spaces, or the surrounding five spaces at the sides of the area, and three at the corners. Let a coin be considered content if no fewer than half of its neighbors are of the same type, and considered discontent otherwise. Following this initial setup, each discontent coin is then moved to a space that will let them be content. Note that this movement may cause new coins that were previously content to now be discontent. This process of checking for content/discontent coins, and then moving all marked as discontent is repeated until some stable equilibrium is reached. With enough blank spaces, they can usually all be satisfied.

Schelling investigated this process by hand, moving coins around on a table top and counting; naturally, there were limits to the number of repetitions he could manage, and to the size of the population he was simulating. By simulating the experiment on a computer, you will be able to run through many more repetitions much more quickly.

Part 1: Representing the grid

We will use two classes to represent the coins in our grid. First, each coin will be represented with the use of a Coin class. This class should have the following:

- An initializer method `__init__(self, type)` that accepts the coin’s type as an int (1 for a penny, 5 for a nickel), stores the type in a data attribute (named `type`), and initializes another attribute named `content` to be `True`.
- An accessor method `getType(self)` to access the coin’s type.
- accessor and mutator methods for the content attribute (`getContent(self)`, `make_content(self)`)

- A `__str__(self)` method that returns a string representation of the coin (e.g., “5C” for a content nickel or “1D” for a discontent penny).

Our second class will represent the grid, itself. Your Grid class should have the following:

- An initializer method `__init__(self)` that creates the grid as a list of lists. The outer list should be called `coins`. It should contain 13 nested lists, each one representing a row of the grid. Each of these nested lists will hold 16 items. Each index of each nested list should contain either an instance of a Coin object or the value `None` (representing an empty space in the grid).
- This `__init__(self)` function should assign 58 spaces in the grid the value `None` (empty spaces), 75 spaces to an instance of a Coin object with type 1 (pennies), and 75 spaces to an instance of a Coin object with type 5 (nickels). These assignments should be random: for each space in the grid, the initializer should randomly select from the available types what to place there (`None`, an nickel, or a penny, as long as there are less than 58 `None`'s in the grid, less than 75 nickels, and less than 75 pennies). If 75 nickels have already been placed, all future spaces should only randomly choose between `None` and pennies (and similarly, if all 75 pennies have been places, only options left are `None` and nickels, etc.).
- A `__str__(self)` method that returns a string formatted to represent the Graph. It should be 13 lines, where each line has 16 entries. Entries on a line should be separated by a single space. Each entry should either be the result of calling `__str__()` on a Coin object or “ ” if that entry has the value `None`.

Note that you will be adding more methods to the Grid class in later parts of the project.

To complete Part 1, you should write a `main()` function that creates a new Grid object and prints it to the screen. Because your Coin objects (and empty spaces) are randomly placed, your program should produce a different grid each time it is run.

Part 2: Finding discontent coins

At this point, each grid that your program produces will be full of seemingly content coins. Your Coin objects are initialized to be content and so far you have not implemented any code that would change that. For Part 2, you will implement a method `check_content(self)` on the Grid class. This function should consider each coin object in the grid one at a time and check if it is content or discontent. After making this determination, it should set each Coin object's content attribute accordingly.

You should consider a coin to be content if 50% or more of its neighbors are of the same type. For each Coin object, you may have to check up to 8 neighbors. Consider a coin in column `j` of row `i` (e.g., the object stored in `coins[i][j]`). You may need to check each of the following:

```
coins[i - 1][j - 1]
coins[i - 1][j]
coins[i - 1][j + 1]
coins[i][j - 1]
coins[i][j + 1]
coins[i + 1][j - 1]
coins[i + 1][j]
coins[i + 1][j + 1]
```

For the coin stored in column 5 of row 4, this would mean checking the following indices:

```
coins[4][3]
coins[4][4]
coins[4][5]
coins[5][3]
coins[5][5]
```

```
coins[6][3]
coins[6][3]
coins[6][4]
coins[6][5]
```

Further note that not all coins will have 8 possible neighbors. the coin stored in the first column of the first row (which, because lists are indexed from 0 would be `coins[0][0]`), only has 3 neighbors: `coins[0][1]`, `coins[1][1]`, and `coins[1][0]`. Hence, this coin would be discontent if any 2 of its neighbors were of the opposite type.

Note that for this part of the project, you should not be moving any discontent coins, yet. You should only check the status of each coin in the grid (content or discontent).

After implementing `check_content(self)`, modify your `main()` function to call `check_content(self)` after initializing the Grid object, but before printing it.

Part 3: Moving discontent coins

Now you will implement a method `move_discontent(self)` on the Grid class to move each of the Coin objects in the grid that are marked as discontent to empty spaces (indices in the nested list data structure that hold a `None` value) where they would be content. Once that Coin object is moved, its `content` attribute should be set to `True` using the `make_content` method. You may move each discontent coin to any space in the grid that is currently empty and would allow the coin to be content. You have the freedom to decide *how* `move_discontent(self)` accomplishes this.

Note that since you are only setting the `content` attribute of the moved coins, other coins in the grid may become discontent as a result of the move. Hence, after a call to `move_discontent(self)`, you will need to call `check_content(self)` again to ensure that the state of the grid is accurately reflected.

Once you have implemented `move_discontent(self)`, modify your `main()` function to allow the user to perform repeated moves. Your main function should:

1. Initialize a Grid object
2. Ensure that `check_content(self)` is called on that object
3. Print the Grid object
4. Ask if the user would like to move discontent pieces (by entering “m”) or quit (by entering “q”). Any other input should be considered invalid.
5. If the user selected to move discontent coins, call `move_discontent(self)` on the Grid object, ensure that the `check_content(self)` is called on the Grid object again, print it again, and prompt the user again.
6. If the user selected to quit, exit the program

Part 4: Processing more iterations

Note that at this point, you are not moving much faster Schelling with his original coins-on-table model. To speed things up, your program should allow the user to instruct your program to process many repetitions without stopping. In addition to accepting “m” and “q” at each prompt, if the user enters an integer between 1 and 1000, your program should process that many iterations, and then display the grid and prompt again (note that this means that entering 1 or entering “m” would do the same thing). Any integers entered outside of this range should be considered invalid.

Once you have completed all parts of the lab, be sure to show your work to the lab instructor

Rubric

Your program will be evaluated according to the following rubric:

Coin class written as specified	10
Grid's <code>__init__</code> method works as specified	15
Grid's <code>__str__</code> method works as specified	5
Grid's <code>check_content</code> method works as specified	20
Grid's <code>move_discontent</code> method works as specified	20
Has a main function and interactive run of the simulation works as specified	20
Code is well laid-out and commented	10